L3 – Deeper Dive on ISAs and LC2k

EECS 370 – Introduction to Computer Organization – Winter 2022

L3_1 ISAs – Instructions and Memory

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

- Identify the addressing modes of memory operations used in assembly-language instructions and programs
- Understand encoding of addressing for assembly-language instructions for load, store, and branching instructions
- Usage and encoding of labels for assembly-language programs

Resources

- Many resources on 370 website
  - https://eecs370.github.io/#resources
    - ARMv8 references
    - Binary, Hex, and 2's compliment
    - Debugging tutorial and reference card
- Discussion resources
- Piazza
- Office hours

The definition of ① bit (smallest unit)
② byte (8 bits)
③ word (32-64 bits) (4-8 bytes)

What is a Bit?

- Bit: Smallest unit of data storage
  - Values `[0, 1]`
  - Many things will be measured (for size) in bits
    - 32-bit register – a register with 32 binary digits of storage capacity
    - 32-bit instruction – machine code instruction that has 32 binary digits, i.e., an unsigned integer in the range `0` to `2^32` (`0` to `4,294,967,296`)
    - 32-bit address – memory addresses with 32 binary digits
    - 32-bit operating system - computer with 32-bit addresses
- Byte: A collection of 8 bits (contiguous)
  - On many computers, the granularity for addresses
- Word: natural group of access in a computer
  - Usually, 32 or 64 bits
  - Useful because most data exceeds 1 byte of storage need

① The location of the data & Instruction code
② How to transfer from Assembly code → machine

Review! Example ISA

Assembly and Machine Code

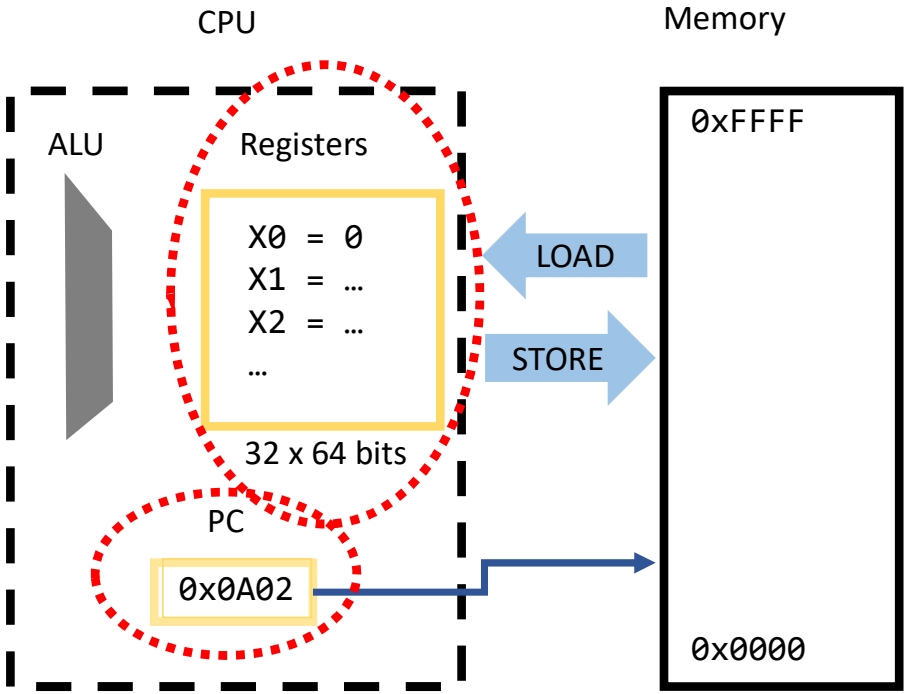
- von Neumann architecture: computers store data and instructions in the same memory
- Instructions *are* data, *encoded* as a **number**

	opcode	dest	src1	src2
Assembly code	ADD	X2	X3	X1
Machine code	011011	010	011	001

Registers

Review! Example ISA

- Registers
  - Small array of storage locations in the processor – general purpose registers
    - Part of the processor – fast to access
    - Direct addressing only
      - That means they can not be accessed by an offset from another address
    - Special purpose registers
      - Examples: program counter (PC), instruction register (IR)



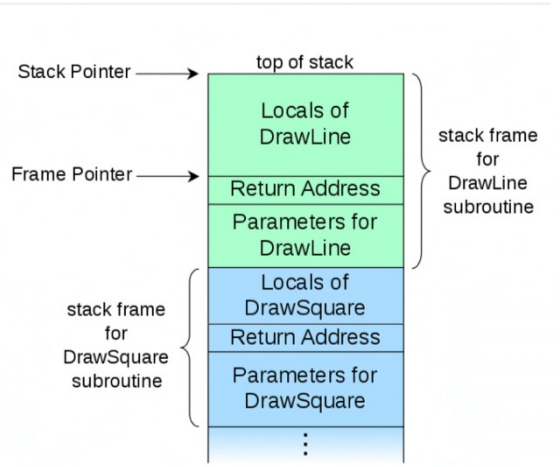
Registers

- ARMv8
  - We will use LEGv8 from Patterson & Hennessy textbook
  - 32 registers, `X0` through `X31`
  - 64-bit wide (64 bits of storage for each register)
  - Some have special uses, e.g., `X31` always contains the value `0`

- LC-2K
  - Architecture used in course projects
  - 8 registers, 32 bits wide each

LC2K is same as LC-2K
Appears both ways in documents in 370

Special Purpose Registers



- Return address
  - Example: ARM register X30, also known as Link Register (LR)
  - Holds the return address or link address of a subroutine
- Stack pointer
  - Examples: ARM register X28 – SP, or x86 ESP
  - Holds the memory address of the stack
- Frame pointer
  - Example: ARM register X29 – FP
  - Holds the memory address of the start of the stack frame
- Program counter (usually referred to as PC)
  - Cannot be accessed directly in most architectures
  - Accessed indirectly through jump insts

When you call a function, you need to return after you finish the function.

stack goes down in the memory, getting larger, as we do deeper function call

(top of the stack)

(point to the local variable in the current function)

These registers store memory addresses

Special Purpose Registers

- 0 value register (ARM register X31 – XZR)
  - no storage, reading always returns 0
  - lots of uses – ex: mov→add

if you write 100 to it and then read from it, you get 0. (throw away the thing you write to it) only return 0

ARM Don't have move instruction: move X_2, X_3 → add X_2 X_3 X_{31}

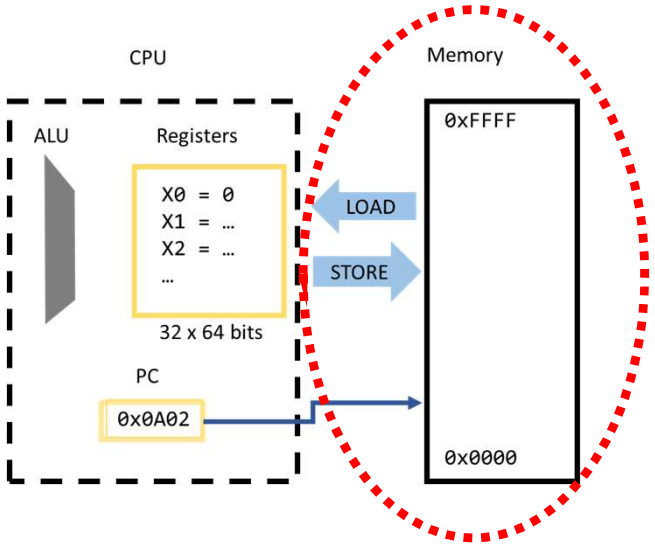
- Status register

tell you the last computation the system make.

  - Examples: ARM SPSR, or x86 EFLAGs
  - Status bits set by various instructions
    - Compare, add (overflow and carry) etc.
  - Used by other instructions like conditional branches

eg: last computation result is negative → use this information to do decisions.

Memory Storage

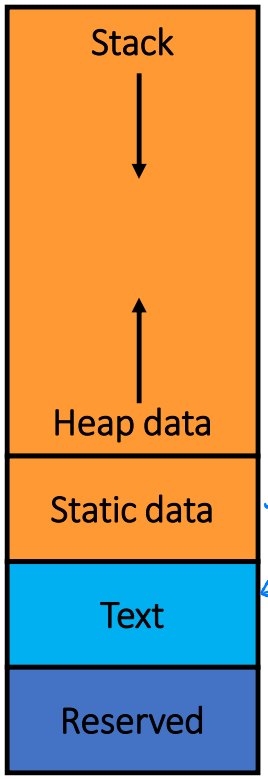


- Large array of storage accessed using memory addresses
- A machine with a 32-bit address can reference memory locations 0 to $2^{32}-1$ (or 4,294,967,295).
- A machine with a 64-bit address can reference memory locations 0 to $2^{64}-1$ (or 18,446,744,073,709,551,615—18 exa-locations)
  - In practice 64-bit machines do not have 64-bit physical addresses

Assembly instructions have multiple ways to access memory (i.e., addressing)

Memory: ARM (Linux) Memory Image

Review!



Activation records: local variables, parameters, etc.

Dynamically allocated data—`new` or `malloc()`

Global data and static local data

Machine code instructions (and some constants)

Reserved for operating system

Addressing Modes

- Addressing (accessing memory using addresses) modes for assembly instructions
  - Direct addressing – memory address is in the instruction
  - Register indirect – memory address is stored in a register
  - Base + displacement – register indirect plus an immediate value
  - PC-relative – base + displacement using the PC special-purpose register

No one use

Direct Addressing

Example ISA (Simplified)

- Specify the address as immediate (constant) in the instruction

load instruction: take a piece of memory, and put it into a register

```
load  r1, M[ 1500 ]    ; r1 <- contents of location 1500
jump #6000             ; jump to address 6000
```

register

location 1500.

the location number is directly stored in the instruction.

Direct Addressing

Example ISA (Simplified)

- Specify the address as immediate (constant) in the instruction

```
load  r1, M[ 1500 ]   ; r1 <- contents of location 1500
jump #6000            ; jump to address 6000
```

- Not practical for something like ARMv8

```
load  r1, M[1073741823]     // 1073741823 is the address in memory
```

With 32-bit instruction encodings, a 32-bit address would fill the instruction!

For ARM: How do you fit a 64-bit address to 32-bit instruction? Can't.

So no direct Address in ARM.

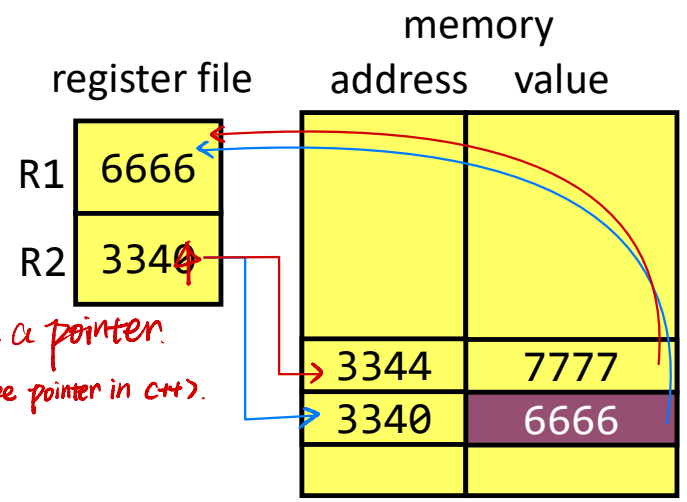
Register Indirect

Example ISA (Simplified)

- Store reference address in a register

use 5 bits to store the register. 64-bit register can store 64-bit address.

```
load r1, M[ r2 ]
add  r2, r2, #4
load r1, M[ r2 ]
```



Now, the register no longer holds a data value, it holds a pointer (like pointer in C++).

Useful for pointers and arrays

`load r1, M[ r2 ]` is a pointer dereference in assembly

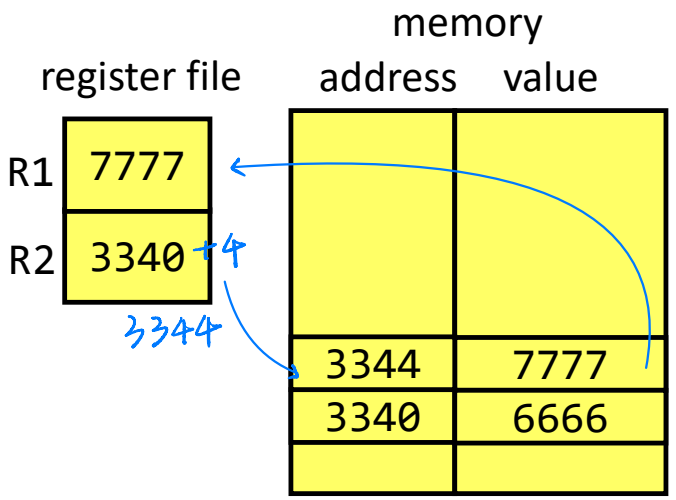
Base + Displacement

Example ISA (Simplified)

the instruction specify a register that contains an address to the start of an object.

- Most common addressing mode
- Address is computed as register value + immediate

```
load r1, M[ r2 + 4 ]
```



Useful for accessing structures or class objects

C code

```
struct Distance {
  int feet;
  int inch;
} x, y;
```

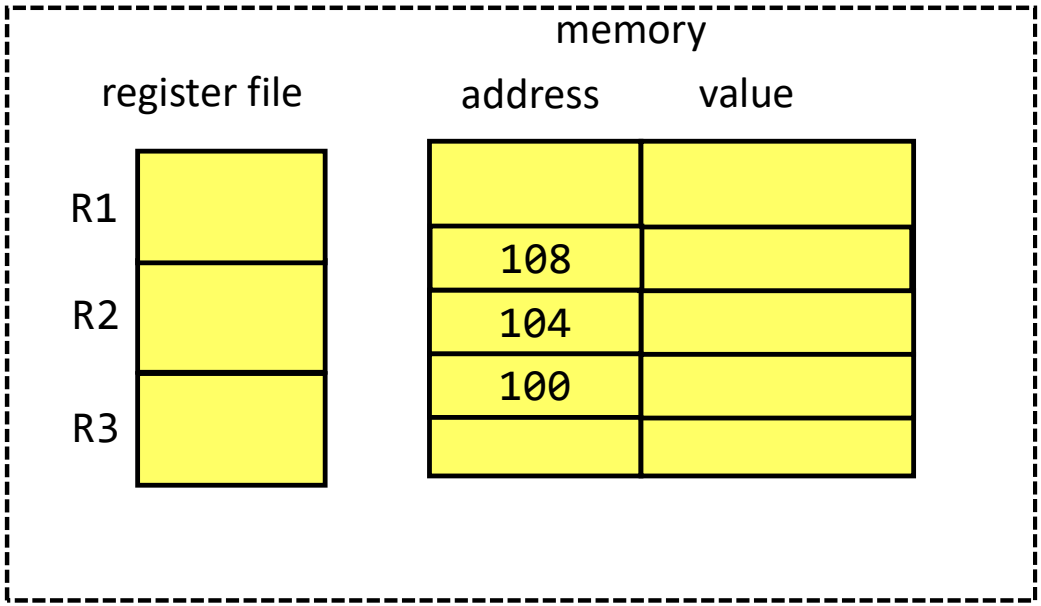
```
x.feet = 11;
y.inches = 5;
```

Addressing Example 1

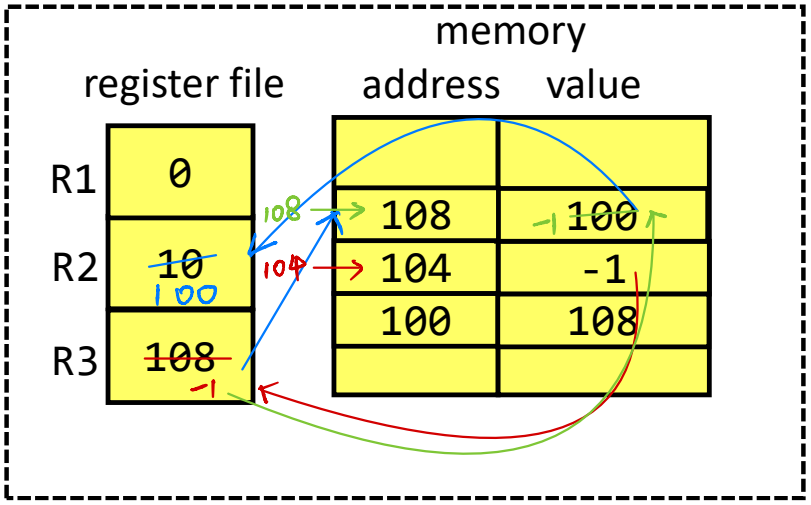
What are the contents of registers and memory after executing the assembly instructions?

Example ISA (Simplified)

```
load  r2, M[ r3 ]
load  r3, M[ r2 + 4 ]
store r3, M[ r2 + 8 ]
```



Starting values



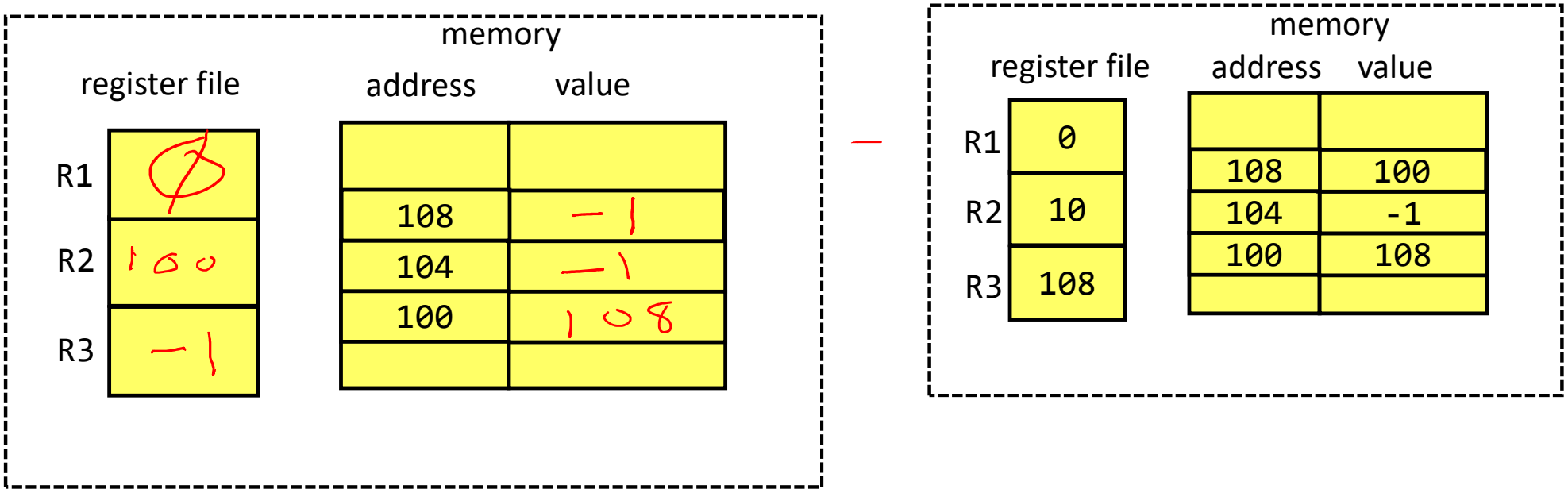
Addressing Example 1

What are the contents of registers and memory after executing the assembly instructions?

Example ISA (Simplified)

```
load  r2, M[ r3 ]
load  r3, M[ r2 + 4 ]
store r3, M[ r2 + 8 ]
```

Starting values



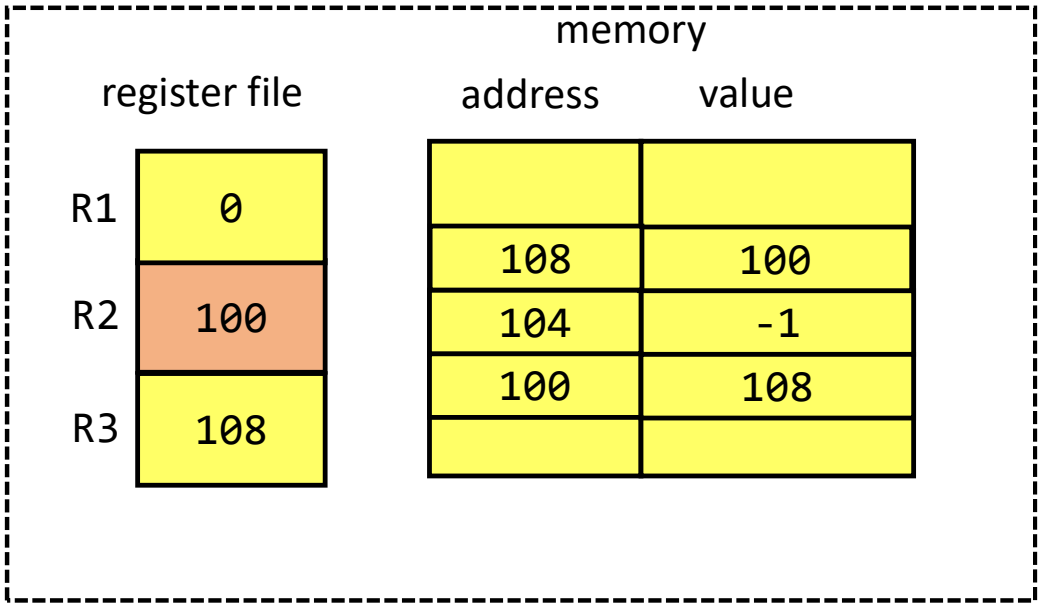
Addressing Example 1

What are the contents of registers and memory after executing the assembly instructions?

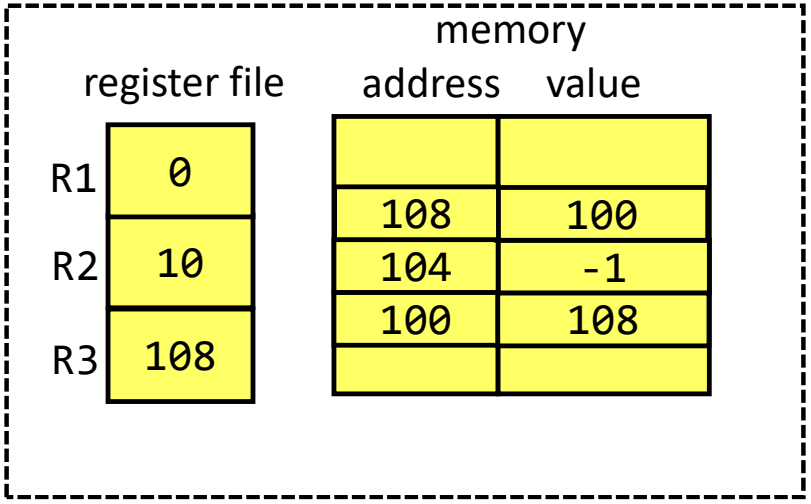
Example ISA (Simplified)

```
load  r2, M[ r3 ]
load  r3, M[ r2 + 4 ]
store r3, M[ r2 + 8 ]
```

```
load  r2, M[ r3 ]
```



Starting values



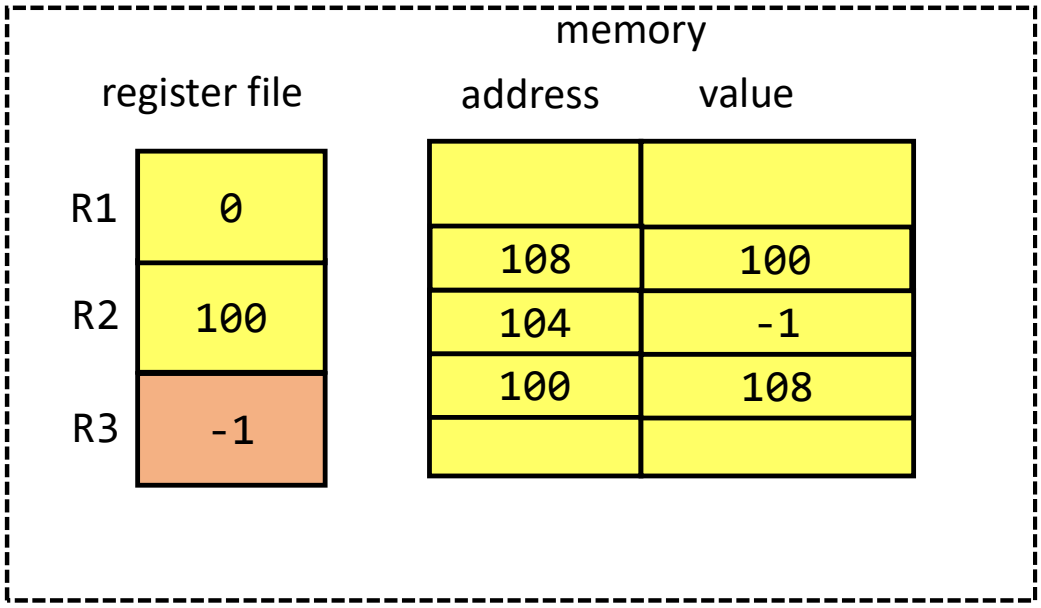
Addressing Example 1

What are the contents of registers and memory after executing the assembly instructions?

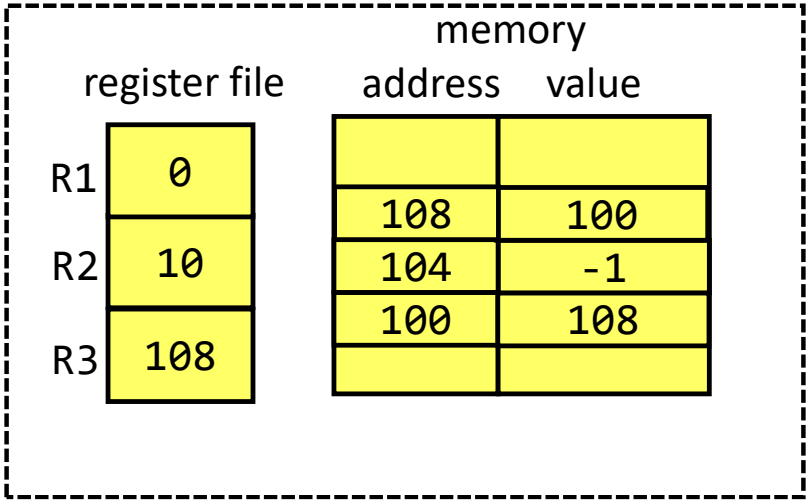
Example ISA (Simplified)

```
load  r2, M[ r3 ]
load  r3, M[ r2 + 4 ]
store r3, M[ r2 + 8 ]
```

```
load  r3, M[ r2 + 4 ]
```



Starting values



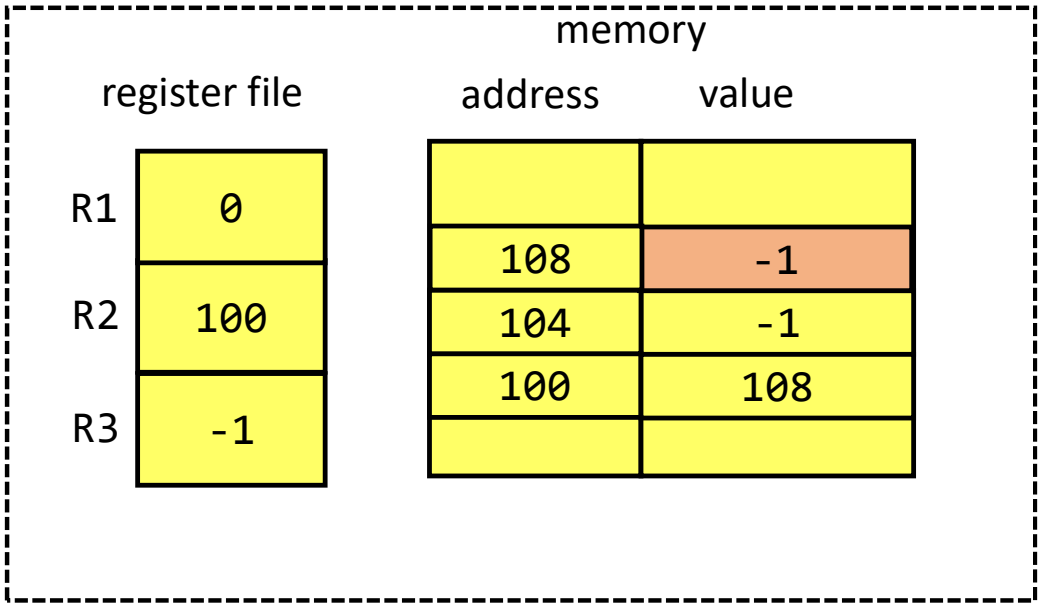
Addressing Example 1

What are the contents of registers and memory after executing the assembly instructions?

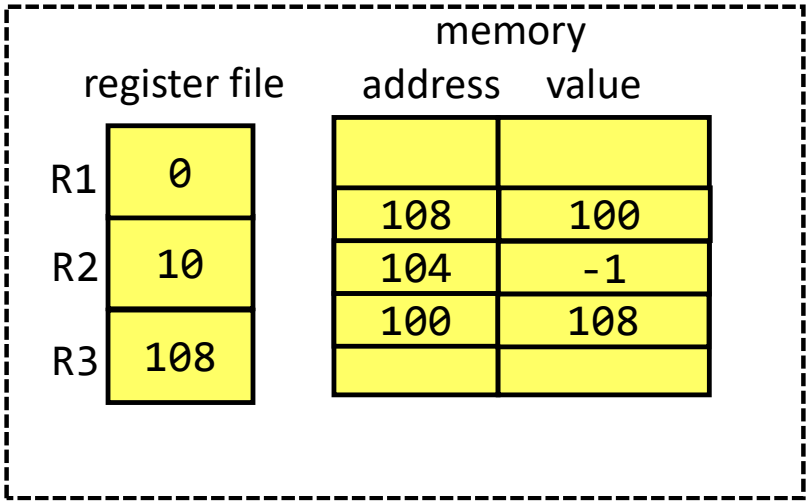
Example ISA (Simplified)

```
load  r2, M[ r3 ]
load  r3, M[ r2 + 4 ]
store r3, M[ r2 + 8 ]
```

```
store r3, M[ r2 + 8 ]
```



Starting values



Program Counter (PC) Relative

Example ISA (Simplified)

- Useful for project - P1a
- Variation of base + displacement
- PC register is the base

Useful for branch instructions!

Relative distance from PC can be positive or negative

eg: loop: you need to jump back.

```
jump   [ -8 ]   jump back 8 bits

jump   [ 2 ]
```

PC-Relative Addressing

LC-2K ISA

- Machine language instructions (encoded from an assembler) use numbers for pc-relative addressing'
- Assembly language instructions (written by people) use ***labels***

PC-Relative Addressing

LC-2K ISA

- Machine language instructions (encoded from an assembler) use numbers for pc-relative addressing'
- Assembly language instructions (written by people) use ***labels***

```
Address
      0          lw 0 1 five     load reg1 with 5 (symbolic address)
      1          lw 1 2 3        load reg2 with -1 (numeric address)
target (2)  start  add 1 2 1      decrement reg1
      3          beq 0 1 2       goto end of program when reg1==0
PC    (4)        beq 0 0 start   go back to the beginning of the loop
      5          noop
      6   done   halt            end of program
      7   five   .fill 5
      8   neg1   .fill -1
      9   stAddr .fill start     will contain the address of start (2)
```

disp (-3)

How to calculate PC relative address:

PC + 1 + displacement = target.

4 + 1 + displacement = 2.

displacement = -3

} Here we want to calculate the displacement.

PC-Relative Addressing

LC-2K ISA

```
Address
      0          lw 0 1 five    load reg1 with 5 (symbolic address)
      1          lw 1 2 3       load reg2 with -1 (numeric address)
      2   start  add 1 2 1      decrement reg1
      3          beq 0 1 2      goto end of program when reg1==0
      4          beq 0 0 start  go back to the beginning of the loop
      5          noop
      6   done   halt           end of program
      7   five   .fill 5
      8   neg1   .fill -1
      9   stAddr .fill start    will contain the address of start (2)
```

PC-Relative Addressing

LC-2K ISA

```
Address
      0           lw 0 1 five     load reg1 with 5 (symbolic address)
      1           lw 1 2 3        load reg2 with -1 (numeric address)
      2   start   add 1 2 1       decrement reg1
      3           beq 0 1 2       goto end of program when reg1==0
      4           beq 0 0 start   go back to the beginning of the loop
      5           noop
      6   done    halt            end of program
      7   five    .fill 5
      8   neg1    .fill -1
      9   stAddr  .fill start     will contain the address of start (2)
```

Project P1a

- After reading specification, downloading starter files, creating project…

- Write test cases to verify your C code
  - Test cases written in LC-2K assembly
- Recommended for a start:

```
t0.ac: halt
t1.ac: noop
       halt
t2.ac: add 1 2 3
       halt
```

```
t3.ac: nor 3 1 4
       halt
```

L3_2 Two's Complement

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

- Represent signed and unsigned numbers in binary (base 2)
- Negate positive and negative signed values
- Complete arithmetic operations (addition and subtraction) by hand using signed and unsigned binary numbers

Binary Review

- Important: get comfortable with representing numbers in binary
- Resources:
  - Video reviews (EECS 370 website): https://drive.google.com/drive/u/3/folders/1T2RIzsHSuQMGtE3eXXGZp9_a4lcyY4ZC
  - Binary review cheat sheet (EECS 370 website): https://eecs370.github.io/resources/number_systems.html

Binary Addition

Review!

- We can already represent non-negative numbers in binary

$$6 \text{ (base 10)} = 2^2 (4) + 2^1 (2) = 110 \text{ (base 2)}$$

- We can do arithmetic with binary numbers

3 + 2 = 5 (base 10)	3 + 5 = 8 (base 10)

Binary Addition

Review!

- We can already represent non-negative numbers in binary

$$6 \text{ (base 10)} = 2^2 \text{ (4)} + 2^1 \text{ (2)} = 110 \text{ (base 2)}$$

4 2 1

- We can do arithmetic with binary numbers

3 + 2 = 5 (base 10)

```
  0 0 1 1
  0 0 1 0
---------
  0 1 0 1
```

3 + 5 = 8 (base 10)

```
  0 0 1 1
  0 1 0 1
---------
  1 0 0 0
```

What about Negative Numbers?

- Thoughts: add another bit for sign, use one of the existing bits for sign

What about Negative Numbers?

- Thoughts: add another bit for sign, use one of the existing bits for sign

1 2

1: 5 = 0101 1 0101 -5 = 0 0101

sign bit

2: 0 101

sign bit

What about Negative Numbers?

- Design space preferences:
  - Representation of positive and negative values
  - Representation of signed and unsigned values
  - Single way to represent 0
  - Equal magnitude of positive and negative values (roughly)
  - Simple (not complex) to detect sign (positive or negative)
  - Simple negation of a number
  - Simple storage for signed and unsigned
  - Simple, non-redundant hardware for operations
    - E.g., one hardware addition unit for signed and unsigned numbers

What about Negative Numbers?

- Design space preferences:
  - Representation of positive and negative values
  - Representation of signed and unsigned values
  - Single way to represent 0
  - Equal magnitude of positive and negative values (roughly)
  - Simple (not complex) to detect sign (positive or negative)
  - Simple negation of a number
  - Simple storage for signed and unsigned
  - Simple, non-redundant hardware for operations
    - E.g., one hardware addition unit for signed and unsigned numbers

- Thought: use existing bit of binary number for signed values Two's Complement

Unsigned Binary Representation

- 1011 in binary is 13 in decimal

$$\begin{array}{cccc} 1 & 1 & 0 & 1 \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array} = 8 + 4 + 1 = 13$$

Unsigned Binary Representation

- 1011 in binary is 13 in decimal

1	1	0	1	=> 8 + 4 + 1 = 13
2^3	2^2	2^1	2^0	

Two's Complement Binary Representation

- 1011 in binary is 13 in decimal

$$1 \quad 1 \quad 0 \quad 1 = 8 + 4 + 1 = 13$$
$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

- Two's complement numbers are very similar to unsigned binary

EXCEPT the first (most significant) digit is negative in two's complement

$$1 \quad 1 \quad 0 \quad 1 = -8 + 4 + 1 = -3$$
$$-(2^3) \quad 2^2 \quad 2^1 \quad 2^0$$

Two's Complement Binary Representation

- 1011 in binary is 13 in decimal

$$1 \quad 1 \quad 0 \quad 1 = 8 + 4 + 1 = 13$$

$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

- Two's complement numbers are very similar to unsigned binary

EXCEPT the first (most significant) digit is negative in two's complement

$$1 \quad 1 \quad 0 \quad 1 = -8 + 4 + 1 = -3$$

$$-(2^3) \quad 2^2 \quad 2^1 \quad 2^0$$

Two's Complement – Exercise 1

What is 1010 (binary)

1. Decimal unsigned value?
2. Decimal signed (two's complement) value?

unsigned

1	0	1	0
2^3	2^2	2^1	2^0

Signed – 2's complement

1	0	1	0
$-(2^3)$	2^2	2^1	2^0

Two's Complement – Exercise 1

What is 1010 (binary)

1. Decimal unsigned value?
2. Decimal signed (two's complement) value?

4 BITS

unsigned

1 0 1 0

2^3 2^2 2^1 2^0

$1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

$8 + 0 + 2 + 0$

$= 10$

Signed – 2's complement

1 0 1 0

$-(2^3)$ 2^2 2^1 2^0

$-8 + 0 + 2 + 0$

-6

Two's Complement – Exercise 1

What is 1010 (binary)

1. Decimal unsigned value?
2. Decimal signed (two's complement) value?

unsigned

$$1 \quad 0 \quad 1 \quad 0$$

$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

$$8 + 2 = 10$$

Signed – 2's complement

$$1 \quad 0 \quad 1 \quad 0$$

$$-(2^3) \quad 2^2 \quad 2^1 \quad 2^0$$

$$-8 + 2 = -6$$

Two's Complement Range

- What is the range of representation of a 4-bit 2's complement number?

  1000 ⟷ 0111

  - [-8, 7] -8 7
- What is the range of representation of an n-bit 2's complement number?
  - $[\ -2^{(n-1)}\ ,\ 2^{(n-1)}\ -1\ ]$

since we both have positive and negative number

Two's Complement Range

- What is the range of representation of a 4-bit 2's complement number?
  - [-8, 7]
- What is the range of representation of an n-bit 2's complement number?
  - [$-2^{(n-1)}$, $2^{(n-1)} - 1$]

32 bits: $[-(2^{31}) \text{ to } 2^{31}-1]$

Negating Two's Complement

- Useful trick: You can negate a 2's complement number by inverting all the bits and adding 1.

5 (decimal) in binary is 0 1 0 1

① Negate (invert) all bits 1 0 1 0

② Add 1 1 0 1 0

\+ 0 0 0 1

1 0 1 1

$-(2^3)$ 2^2 2^1 2^0

-8 + 0 + 2 + 1 = -5

if negation is fast, we only need add.

eg: 5 - 3 == 5 + (-3)

Two's Complement – Exercise 2

How would you represent -3 (decimal) in 2's complement binary using 4 bits?

Two's Complement – Exercise 2

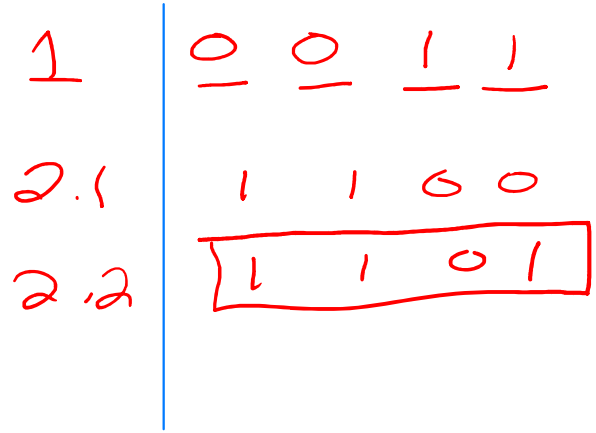
How would you represent -3 (decimal) in 2's complement binary using 4 bits?

1. Convert 3 (decimal) to binary
2. Negate binary
    1. Invert all bits
    2. Add one

Two's Complement – Exercise 2

How would you represent -3 (decimal) in 2's complement binary using 4 bits?

1. Convert 3 (decimal) to binary
2. Negate binary
   1. Invert all bits
   2. Add one



Two's Complement – Exercise 2

How would you represent -3 (decimal) in 2's complement binary using 4 bits?

1. Convert 3 (decimal) to binary
2. Negate binary
    1. Invert all bits
    2. Add one

1. Convert 3 to binary
    1. 3 -> 0011
2. Convert to 2's complement
    1. 0011 -> 1100
    2. 1100 + 1 = 1101

Signed – 2's complement

1	1	0	1
-2^3	2^2	2^1	2^0

$-8 + 4 + 0 + 1 = -3$

Sign Extension

- With two's compliment, it matters how many bits are used!

```
5 (decimal) in binary (4 bits) is   0101
5 (decimal) in binary (8 bits) is   0000 0101

-5 (decimal) in binary (4 bits) is  1011
-5 (decimal) in binary (8 bits) is  1111 1011
```

1+2+0+-8 = -5.

128 64 32 16 8 4 2 1

this is a byte

-128+64+32+16+8+2+1

NOT 0000 1011

-8+2+1 = -5.

need to **extend the most significant (sign) bit**

LC-2K: programmer (you) need to do this!

+++0 1101

↓ (overflow)

0 1101

means you need 6 bits to express the number

Two's Complement Arithmetic

Decimal	2's Complement Binary	Decimal	2's Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

7 – 6 = 7 + (-6) = 1

0111 – 0110
=
0111 + 1010
=
~~1~~0001 4 bits
=
1

6 – 7 = 6 + (-7) = -1

0110 – 0111
=
0110 + 1001
=
1111
=
-1

0110
+ 1001
1111

Two's Complement Arithmetic

Decimal	2's Complement Binary	Decimal	2's Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

7 – 6 = 7 + (-6) = 1

```
  1 0 1 1 1
    1 0 1 0
-------------
1 0 0 0 1
```

6 – 7 = 6 + (-7) = -1

```
  0 1 1 0
    1 0 0 1
-------------
  1 1 1 1
```

L3_3 LC-2K ISA

EECS 370 – Introduction to Computer Organization – Winter 2022

Learning Objectives

- Recognize the set of instructions for LC-2K Architecture (ISA) and be able to describe the operations and operands for each instruction
- Ability to create simple LC-2K assembly programs, e.g., using addition and branching.
- Understand and be able to replicate the encoding (translation from assembly to machine code) of instructions for any LC-2K assembly program

LC-2K Processor

LC-2K ISA

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
- supports 65536 words of memory (addressable space)
- 8 instructions in the following common categories:
  - Arithmetic: add
  - Logical: nor
  - Data transfer: lw, sw
  - Conditional branch: beq
  - Unconditional branch (jump) and link: jalr
  - Other: halt, noop

16 bits address

x nor x = invert.

load word: move piece of memory from 32-bit entry memory to register

store word: take register value and put it to the memory

decision Instruction: compare two registers. to see if their values are equal. if they are equal. they use PC relative addressing. using displacement in the instruction to jump to the target. Otherwise. it go to the next target.

function call tool.

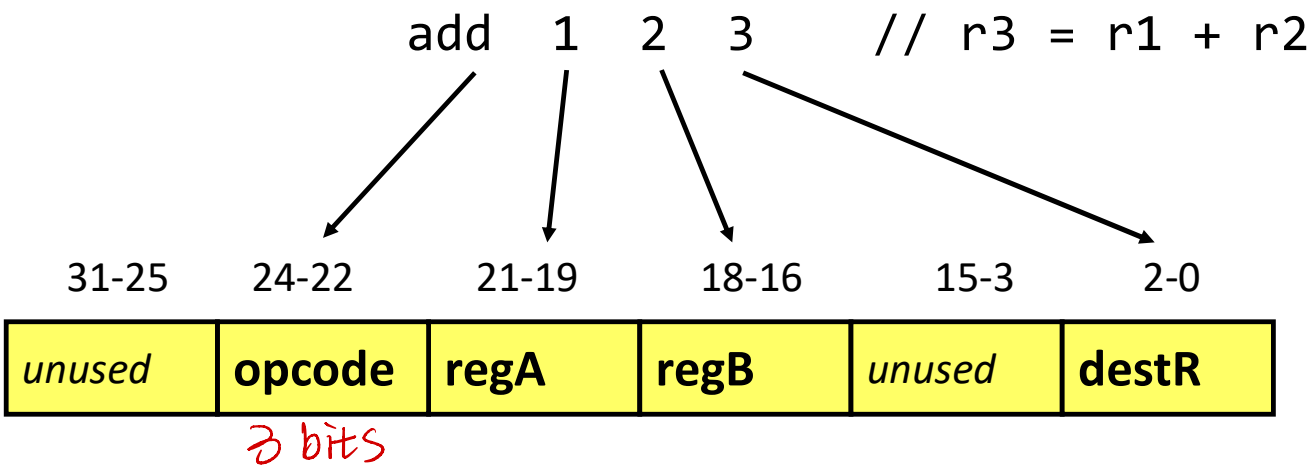
does nothing.

Stop running the program and print out the result.

Instruction Encoding

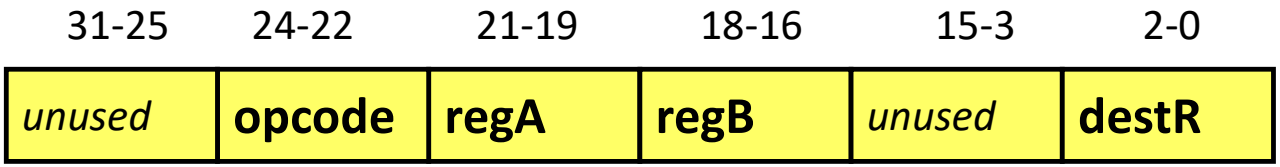
LC-2K ISA

- The Instruction Set Architecture (aka Architecture) defines the mapping of assembly instructions to machine code

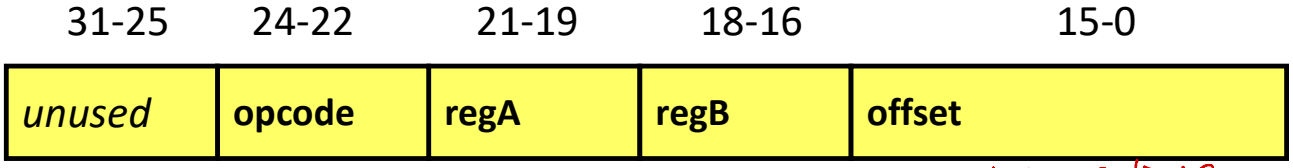


Instruction Formats – R-type, I-type

- Tells you which bit fields correspond to which part of an assembly instruction
- R-type (register) – ① `add` (opcode `000`), ② `nor` (opcode `001`)



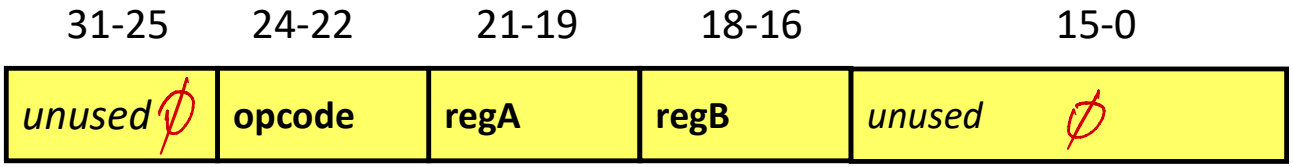
- I-type (immediate) - ① `lw` (opcode `010`), ② `sw` (opcode `011`), ③ `beq` (opcode `100`)



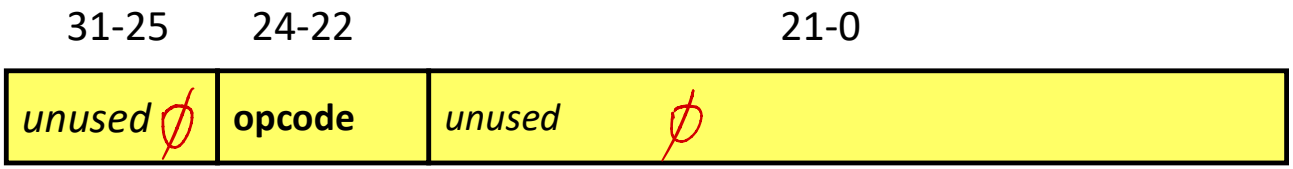
assigned immediate (2's complement)

Instruction Formats – J-type, O-type

- J-type (jump) – `jalr` (opcode `101`



- O-type (???) - `halt` (opcode `110`), `noop` (opcode `111`)



Instruction Formats

- The Instruction Set Architecture (aka Architecture) defines the mapping of assembly instructions to machine code

Instruction Type	Instruction	Bits 31-25	Bits 24-22	Bits 21-19	Bits 18-16	Bits 15-3	Bits 2-0
R-type	add	unused	opcode	reg A	reg B	unused	destReg
	nor						
I-type	lw					offsetField 16-bit, 2's complement number range:[-32768, 32767]	
	sw						
	beq						
J-type	jalr					unused	
O-type	halt			unused			
	noop						

this part will tell you what the rest part does.

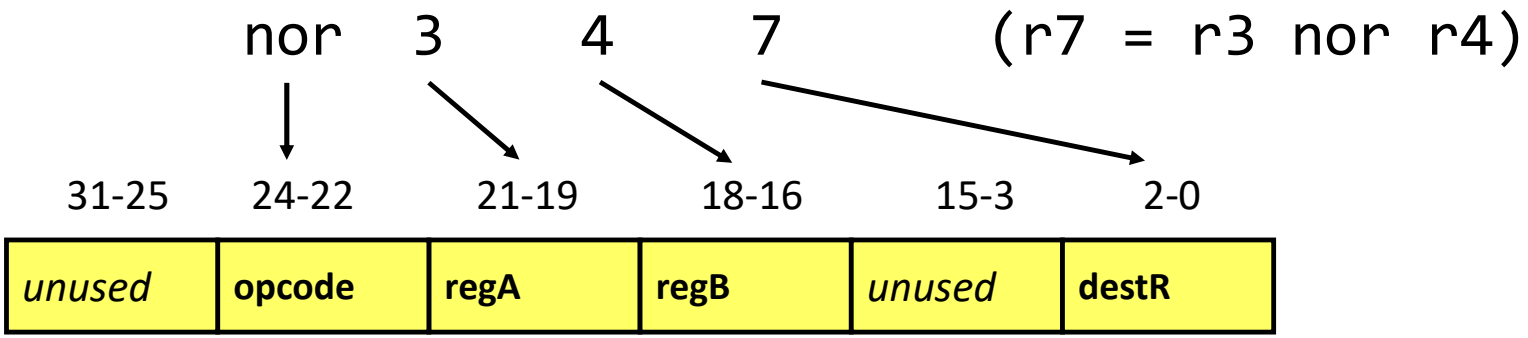
Unused: all unused bits should always be 0

Bit Encodings

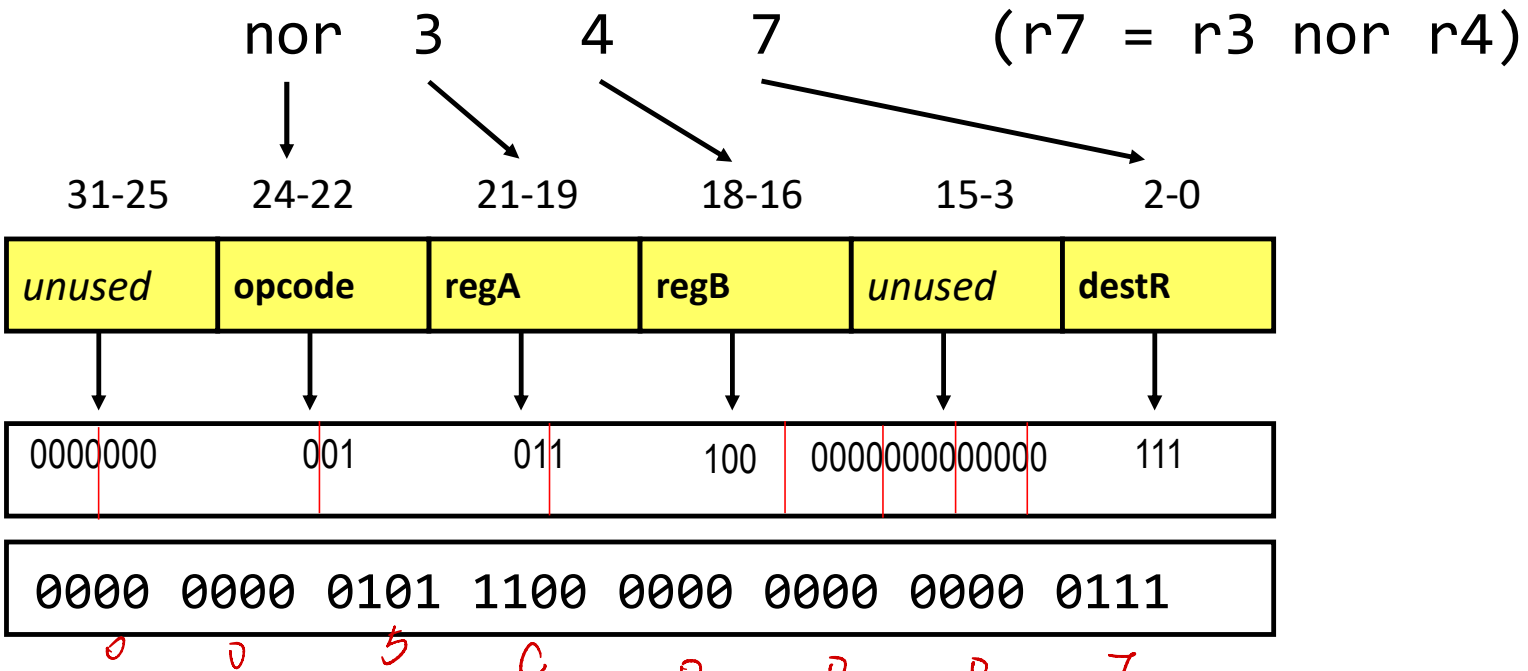
- Opcode binary encodings:
  - `add (000), nor (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)`
- Register operands
  - Binary encoding of register number, e.g., `r2 = 2 = 010`
- Immediate values
  - Binary encoding *using 2's complement values*
  - Give all available bits a value – *do not forget sign extension!*

16 bits

Encoding Example #1 - **nor**



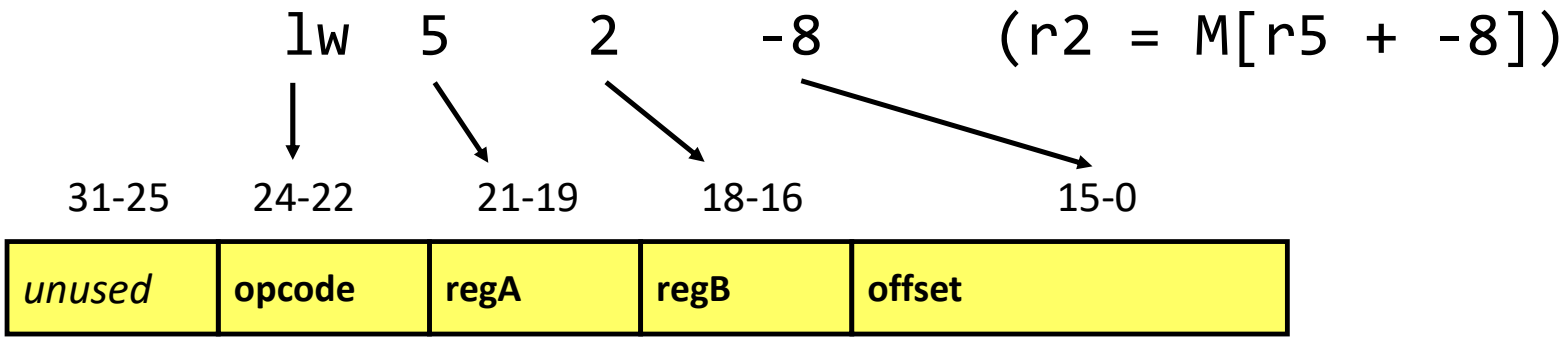
Encoding Example #1 - **nor**



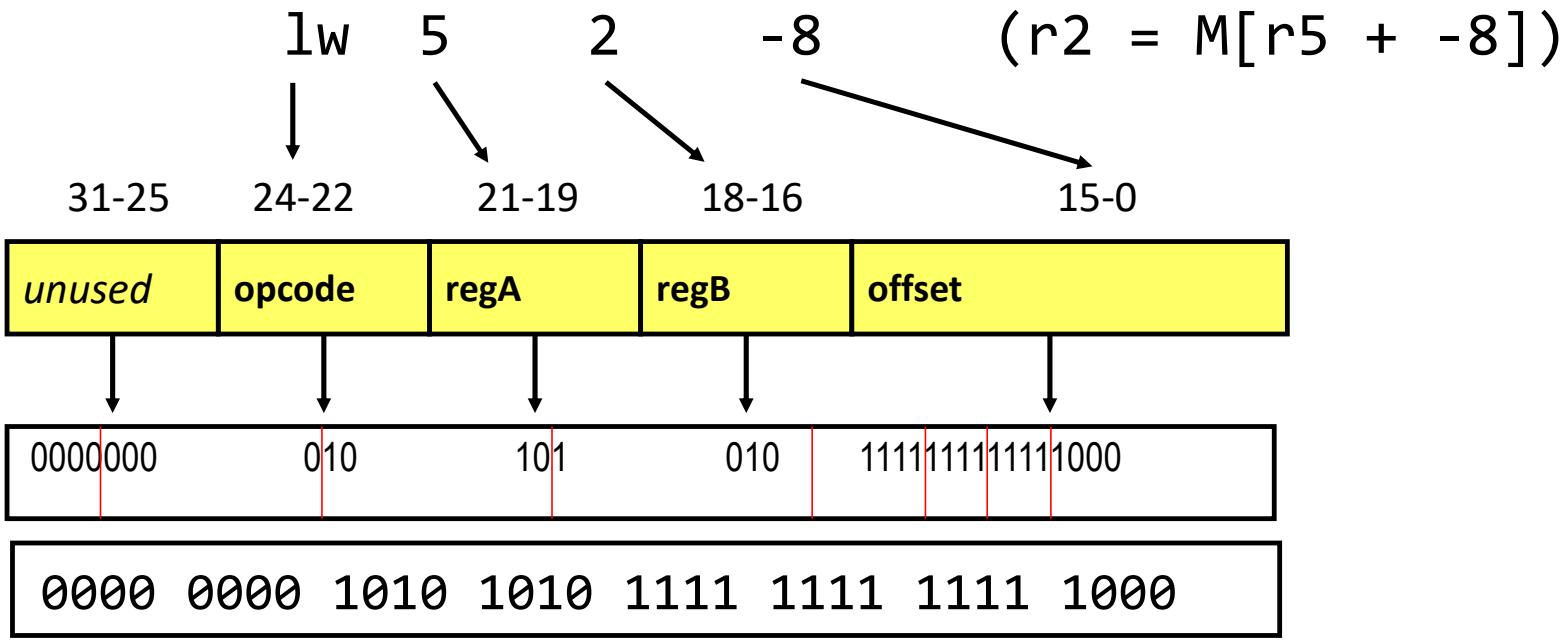
Convert to Hex → 0x005C0007

Convert to Dec → 6029319

Encoding Example #2 - `lw`



Encoding Example #2 - `lw`



Convert to Hex → 0x00AAFFF8
Convert to Dec → 11206648

Encoding Example #3 - add

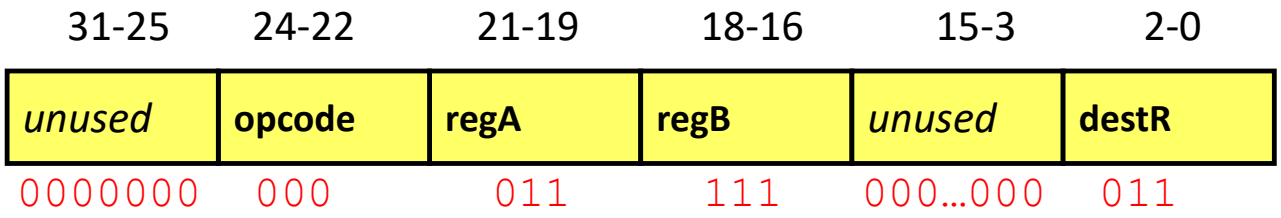
- Compute the encoding in Hex for:

```
add   3    7    3      (r3 = r3 + r7)      (add = 000)
```

Encoding Example #3 - **add**

- Compute the encoding in Hex for:

```
add   3    7    3      (r3 = r3 + r7)      (add = 000)
```

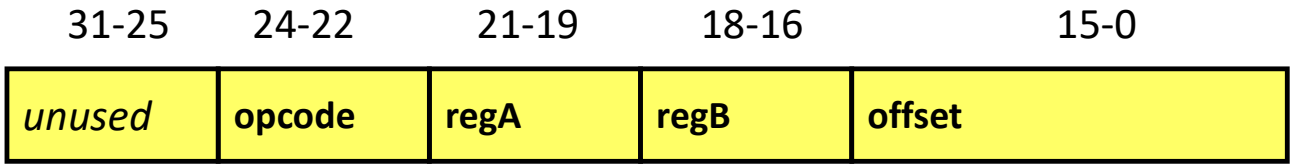


Convert to Hex → 0x001F000003
Convert to Dec → 2031619

Encoding Example #4 - **`sw`**

- Compute the encoding in Hex for:

```
sw   1    5   67      (M[r1+67] = r5)    (sw = 011)
```

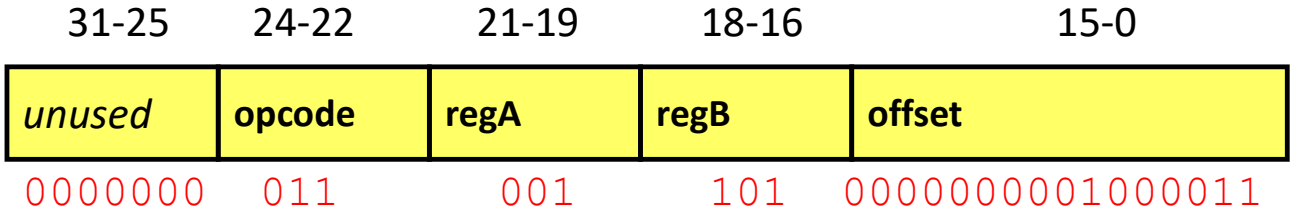


Encoding Example #4 - `sw`

- Compute the encoding in Hex for:

```
sw   1    5   67      (M[r1+67] = r5)    (sw = 011)
```

memory address.



Convert to Hex → 0x00CD0043
Convert to Dec → 13434947

Assembler, aka, P1a

- Each line of assembly code corresponds to a number
  - “add 0 0 0” is just 0.
  - “lw 5 2 -8” is 11206648
- Assembly code is how people write instructions for an ISA
  - We only use assembly because it’s easier to read.
- Assembly code must be assembled (instructions encoded) to machine code for execution

① if start with letter: it's an opcode for an instruction
② if start with dot: it's a command for the assembler

Assembler Directive - `.fill`

汇编指令

- You might want a number to be, well, a number.
  - Data for `lw`, `sw` instructions will be added to LC-2K assembly code file
- `.fill` tells the assembler to put a number instead of an instruction
- The syntax (to have a value of 7) is just `.fill 7`

this is not an instruction, just take the signed number and put it into the memory at current location in the file. ⟶ used to initialize variable.

- Question:
  - What do `.fill 7` and `add 0 0 7` have in common?

Assembler Directive - `.fill`

- You might want a number to be, well, a number.
  - Data for `lw`, `sw` instructions will be added to LC-2K assembly code file
- `.fill` tells the assembler to put a number instead of an instruction
- The syntax (to have a value of 7) is just `.fill 7`
- Question:
  - What do `.fill 7` and `add 0 0 7` have in common?

They have the same value in machine code: 7 (decimal) `111` (binary)
really `0000 0000 0000 0000 0000 0000 0000 0111`

Labels in LC-2K

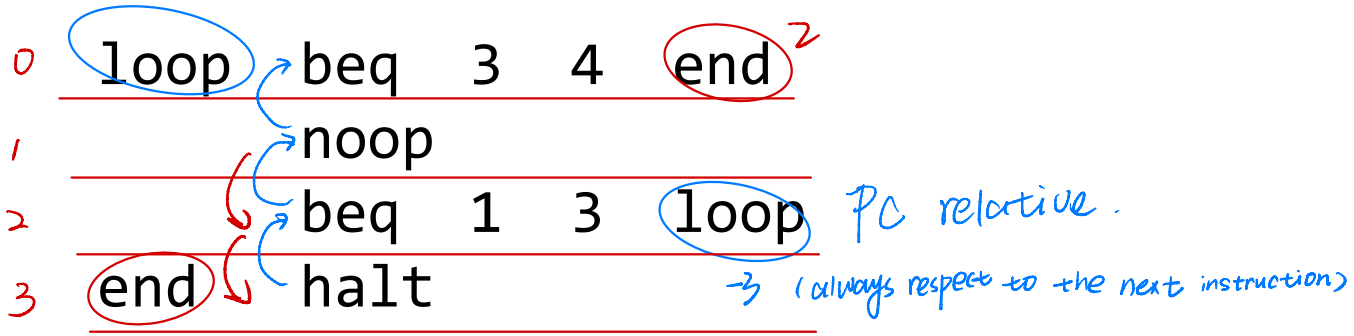
- Labels are used in lw/sw instructions or beq instruction

- For `lw` or `sw` instructions, the assembler should compute `offsetField` to be equal to the address of the label
  - i.e. `offsetField` = address of the label

- For `beq` instructions, the assembler should translate the label into the numeric `offsetField` needed to branch to that label
  - i.e. `PC+1+ offsetField` = address of the label

Labels in LC-2K – Example #1

- Labels are a way of referring to a line in an assembly-language program



Labels in LC-2K – Example #1

- Labels are a way of referring to a line in an assembly-language program

```
loop   beq   3   4   end
       noop
       beq   1   3   loop
end    halt
```

```
loop  is address 0
end is address 3
```

Replacing use of labels with values

```
Addresses   instructions
        0   loop   beq   3   4   2
        1          noop
        2          beq   1   3   -3
        3   end    halt
```

Program in LC-2K – Example

1. Encode program instructions 2. What does this program do?

```
loop  lw   0  1  one
      add  1  1  1
      sw   0  1  one
      halt
one   .fill 1
```

Program in LC-2K – Example

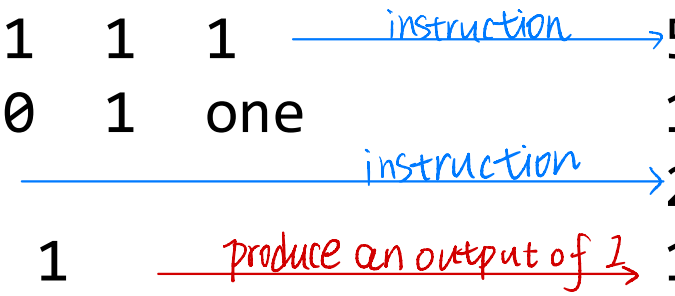
1. Encode program instructions 2. What does this program do?

test.as

```
loop   lw    0   1   one
       add   1   1   1
       sw    0   1   one
       halt
one    .fill 1
```

test.mc

```
8454148
589825
12648452
25165824
1
```



Logistics

- There are two worksheets for lecture 3
  1. Addressing and 2's complement
  2. LC-2K program encoding
- Complete the participation quiz for lecture 3 on Canvas
  - Not graded, but useful to reinforce these concepts!